

Language Parsing

Robert M. Keller
Harvey Mudd College
12 October 2003

Parsing

- Parsing is the process of determining whether a string is in a given language.
- Every compiler and interpreter does some form of parsing.
- Closely related is the process of assigning a meaning to the strings that are in the language.
- Refer to CS 60 materials for a review of how this can be done.



Parsing with a Pushdown Automaton

- We have seen how pushdown automata parse languages represented by context-free grammars.
- In some cases, the corresponding automaton is non-deterministic, which limits the practical use of pda's as a programming technique.
- Only a proper subset of languages can be parsed **deterministically** by a pda. Intuitively, these Deterministic Context-Free Languages are the ones that don't require "guessing".



Use of End-Markers

- In general, a pda will tell us when to accept the string parsed **so far**, as if there might be more coming.
- Sometimes we want to tell the pda that we **don't want an answer** to this question until all input has been supplied.
- One way to do this is to include an **endmarker**, say \$, in the language being defined. The language will have the form
 $\{x\$ \mid x \text{ has some property}\}$
- The pda can react to the endmarker when it sees it.

Deterministic Parsing using Look-ahead

- A non-deterministic pda can sometimes be rendered "deterministic" if there is some way to restrict the choice of moves to at most one.
- The best-known cases are to use "look-ahead":
 - By using **knowledge of the next symbol in the input**, only certain rules (corresponding to certain grammar productions) can be seen to be applicable. The ability to do this depends on having the appropriate grammar. **The knowledge itself is outside the grammar.**
 - The LL(1) grammars can be parsed top-down using produce-match with one symbol look-ahead.
 - The LL(R) grammars can be parsed bottom-up using shift-reduce with one symbol look-ahead.
 - The concept of LL(k) and LR(k) are due to Donald Knuth.

LL(1) grammars


- For each production $A \rightarrow \alpha$ a set of terminal symbols $\text{Lookahead}(A \rightarrow \alpha)$ is **computed**. These are the symbols that could possibly occur **after A** in a sentential form.
- If no two productions with A as the LHS have symbols in common in their Lookahead sets, then the grammar is LL(1).
- In this case, knowledge of $\text{Lookahead}(A \rightarrow \alpha)$ tells us whether to apply production $A \rightarrow \alpha$ in top-down parsing.
- An LL(1) grammar is not necessarily the most natural. It may take a transformation to get an equivalent LL(1) grammar, or there may be none.

Example

- $E \rightarrow T \mid E + T$
- $T \rightarrow F \mid T * F$
- $F \rightarrow a \mid (E)$
- is **not** LL(1), since e.g.
Lookahead($E \rightarrow T$) = {a, (} and
Lookahead($E \rightarrow E + T$) = {a, (}
- So we'd need to transform this grammar.

Example

- $E \rightarrow T E'$
- $E' \rightarrow + T E' \mid \epsilon$
- $T \rightarrow F T'$
- $T' \rightarrow * F T' \mid \epsilon$
- $F \rightarrow a \mid (E)$
- **is** LL(1), e.g.
Lookahead($E' \rightarrow + T E'$) = {+} but
Lookahead($E' \rightarrow \epsilon$) = {), \$}
Lookahead($T' \rightarrow * F T'$) = {*} but
Lookahead($T' \rightarrow \epsilon$) = {), +, \$}



LR(1)

- Handles a broader family of languages than LL(1) can.
- Bottom-up, shift-reduce parsing.
- Difficult to transform.
- Automated tools such as YACC can help.



CYK Algorithm

- CYK = “Cocke, Younger, Kasami”, independent discoverers of the same algorithm.
- Deterministic, not using pda model.
- Does not require much transformation of the grammar.

CYK Motivation

- Problem is to determine whether a given string x is in $L(G)$.
- A **naïve** approach, would be to generate all possible strings, in increasing length, until either:
 - x is generated, **or**
 - all strings of length x or shorter have been generated.
- This is a very **slow** process; could be exponential time in the length of the string (e.g. if each symbol could have been generated by two different productions: $2 \times 2 \times \dots \times 2$ n times).

CYK Motivation

- Rather than working from the start symbol toward generated strings, might be better to work from string backward, to see if start symbol could have generated the string.
- This too could be exponential if not done carefully.
- The CYK uses "dynamic programming" to make the process efficient.

Dynamic Programming?

- Recursive expressions, such as $f(n) = f(n-1) + f(n-3)$; $f(n) = 1$ if $n < 3$; while **very clear** in their intent, may be **inefficient** if taken literally.
- They may entail much **recomputation** unless care is taken to ensure otherwise.
- The idea of dynamic programming is to compute from the "**bottom up**":
 $f(0), f(1), f(2), f(3), \dots$
remembering values for **posterity** even if they might not be used ultimately.

CYK Algorithm

- Let $x = x_1 x_2 \dots x_n$ be the string to be parsed.
- Define $a(i, j) =$
$$\{B \mid B \rightarrow^* x_i x_{i+1} \dots x_j\}$$

for each $i, j \in \{1, 2, \dots, n\}$ with $i \leq j$.
- So $x \in L(G)$ iff $S \in a(1, n)$.
- The essence of CYK is how to compute $a(1, n)$ efficiently.

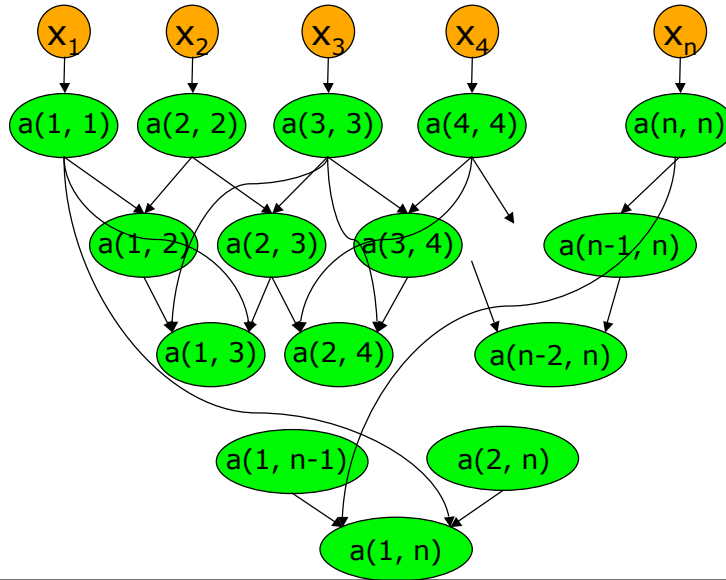
CYK Algorithm

- Assume that the grammar is Chomsky Normal Form.
- We arrive at $a(1, n)$ by the following method:
 - First compute $a(i, i)$ for each i :
$$a(i, i) = \{B \mid B \rightarrow x_i\}$$
- Since the grammar is in CNF, the indicated productions are the only ones that produce terminal symbols.

CYK Algorithm

- Next compute $a(i, i+1)$ for each i , then $a(i, i+2)$ for each i , and so on.
- To compute $a(i, j)$ in general, we use pairs, which have been computed in the previous iteration:
$$\begin{array}{l} a(i, i) \quad a(i+1, j) \\ a(i, i+1) \quad a(i+2, j) \\ a(i, i+2) \quad a(i+3, j) \\ \dots \\ a(i, j-1) \quad a(j, j) \end{array}$$
- For each pair, we need only to check whether there is a production of the form $A \rightarrow BC$ where $B \rightarrow a(i, i+k)$ and $C \rightarrow a(i+k-1, j)$ since the grammar is in CNF.

CYK Algorithm Data Flow



CYK "Wavefront" Matrix

→

a(1, 1)	a(1, 2)	a(1, 3)	a(1, 4)		a(1, n-1)	a(1, n)
	a(2, 2)	a(2, 3)	a(2, 4)		a(2, n-1)	a(2, n)
		a(3, 3)	a(3, 4)		a(3, n-1)	a(3, n)
			a(4, 4)			
					a(n-1, n-1)	a(n-1, n)
						a(n, n)

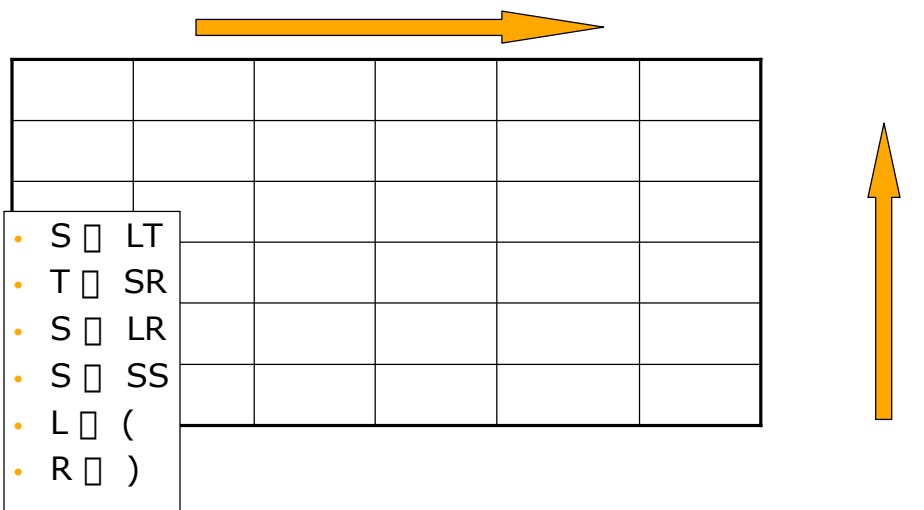
↑

Each entry is computed from entries in its same row and column, e.g. $a(1, 4)$ from $a(1,1)$ and $a(2, 4)$, $a(1, 2)$ and $a(3, 4)$, $a(1, 3)$ and $a(4, 4)$.

CYK Example

- $S \rightarrow LT$
 - $T \rightarrow SR$
 - $S \rightarrow LR$
 - $S \rightarrow SS$
 - $L \rightarrow ($
 - $R \rightarrow)$
- This grammar is in CNF.

CYK Movie on input $((())())$



CYK Movie on input (()())



L					
	L				
		R			
			L		
				R	
					R

- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input (()())



L	\emptyset				
	L	S			
		R	\emptyset		
			L	S	
				R	\emptyset
					R

- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input (()())



L	∅	∅			
	L	S	∅		
		R	∅	∅	
			L	S	T
				R	∅
					R

- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input (()())

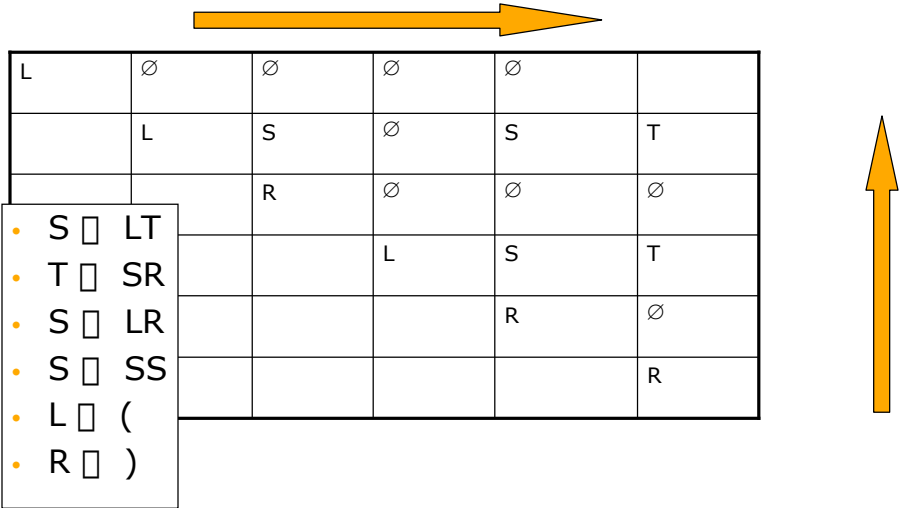


L	∅	∅	∅		
	L	S	∅	S	
		R	∅	∅	∅
			L	S	T
				R	∅
					R

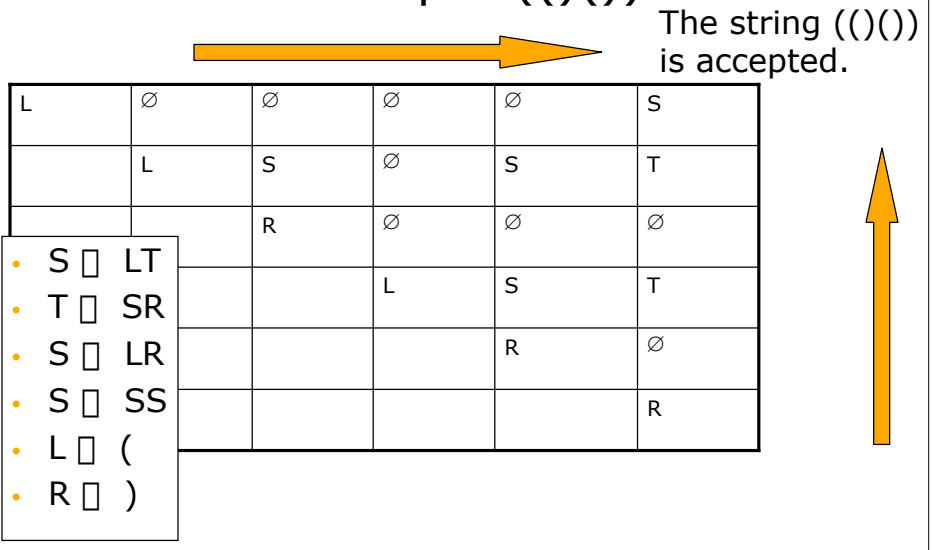
- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input (()())



CYK Movie on input (()())



CYK Movie on input ())(())



- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input ())(())



L	S				
	R	\emptyset			
		R	\emptyset		
			L	S	
				R	\emptyset
					R

- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input ())(())



L	S	T			
	R	∅	∅		
		R	∅	∅	
			L	S	T
				R	∅
					R

- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input ())(())



L	S	T	∅		
	R	∅	∅	∅	
		R	∅	∅	∅
			L	S	T
				R	∅
					R

- S \square LT
- T \square SR
- S \square LR
- S \square SS
- L \square (
- R \square)



CYK Movie on input ())(())

- $S \rightarrow LT$
- $T \rightarrow SR$
- $S \rightarrow LR$
- $S \rightarrow SS$
- $L \rightarrow ($
- $R \rightarrow)$

L	S	T	∅	∅	∅
	R	∅	∅	∅	∅
		R	∅	∅	∅
			L	S	T
				R	∅
					R

CYK Movie on input ())(())

The string ())(()) is not accepted.

- $S \rightarrow LT$
- $T \rightarrow SR$
- $S \rightarrow LR$
- $S \rightarrow SS$
- $L \rightarrow ($
- $R \rightarrow)$

L	S	T	∅	∅	∅
	R	∅	∅	∅	∅
		R	∅	∅	∅
			L	S	T
				R	∅
					R

CYK Movie on input abbaa (different grammar)



- $S \rightarrow AB$
- $S \rightarrow BC$
- $A \rightarrow AB$
- $A \rightarrow a$
- $B \rightarrow AA$
- $B \rightarrow CB$
- $B \rightarrow b$
- $C \rightarrow a$
- $C \rightarrow b$



CYK Movie on input abbaa (different grammar)



- $S \rightarrow AB$
- $S \rightarrow BC$
- $A \rightarrow AB$
- $A \rightarrow a$
- $B \rightarrow AA$
- $B \rightarrow CB$
- $B \rightarrow b$
- $C \rightarrow a$
- $C \rightarrow b$

A, C				
	B, C			
		B, C		
			A, C	
				A, C



CYK Movie on input abbaa (different grammar)



- S \rightarrow AB
- S \rightarrow BC
- A \rightarrow AB
- A \rightarrow a
- B \rightarrow AA
- B \rightarrow CB
- B \rightarrow b
- C \rightarrow a
- C \rightarrow b

A, C	S, A, B			
	B, C	S, B		
		B, C	S	
			A, C	B
				A, C



CYK Movie on input abbaa (different grammar)



- S \rightarrow AB
- S \rightarrow BC
- A \rightarrow AB
- A \rightarrow a
- B \rightarrow AA
- B \rightarrow CB
- B \rightarrow b
- C \rightarrow a
- C \rightarrow b

A, C	S, A, B	S, A, B		
	B, C	S, B	S	
		B, C	S	B
			A, C	B
				A, C



CYK Movie on input abbaa (different grammar)



- $S \rightarrow AB$
- $S \rightarrow BC$
- $A \rightarrow AB$
- $A \rightarrow a$
- $B \rightarrow AA$
- $B \rightarrow CB$
- $B \rightarrow b$
- $C \rightarrow a$
- $C \rightarrow b$

A, C	S, A, B	S, A, B	S, B	
	B, C	S, B	S	B
		B, C	S	B
			A, C	B
				A, C



CYK Movie on input abbaa (different grammar)



The string abbaa is accepted.

- $S \rightarrow AB$
- $S \rightarrow BC$
- $A \rightarrow AB$
- $A \rightarrow a$
- $B \rightarrow AA$
- $B \rightarrow CB$
- $B \rightarrow b$
- $C \rightarrow a$
- $C \rightarrow b$

A, C	S, A, B	S, A, B	S, B	S, A, B
	B, C	S, B	S	B
		B, C	S	B
			A, C	B
				A, C



Complexity of CYK

- n is the length of the input string.
- p is the number of productions, which can be treated as a constant.
- There are $O(n^2)$ sets to be computed.
- Each set can be represented as a **bit-vector**, so that elements can be added and membership-tested in $O(1)$ time.
- A general set can be computed in $O(n)$ iterations, each iteration involving examination of p productions.
- The total time is proportional to $pn^2 \approx O(n^3)$.

Expressing CYK Iteratively

- Work by super-diagonals $d = 2, 3, \dots, n$
- $d = 2$ compute:
 - $a(1, 2) = a(1, 1) \ a(2, 2)$ is pairing
 - $a(2, 3) = a(2, 2) \ a(3, 3)$
 - ...
 - $a(n-1, n) = a(n-1, n-1) \ a(n, n)$
- $d = 3$ compute:
 - $a(1, 3) = a(1, 1) \ a(2, 3) \ \square \ a(1, 2) \ a(3, 3)$
 - $a(2, 4) = a(2, 2) \ a(3, 4) \ \square \ a(2, 3) \ a(4, 4)$
 - ...
 - $a(n-2, n) = a(n-2, n-2) \ a(n-1, n) \ \square \ a(n-2, n-1) \ a(n, n)$
- . . .
- $d = n$ compute:
 - $a(1, n) = a(1, 1) \ a(2, n) \ \square \ a(1, 2) \ a(3, n) \ \square \ \dots$
 $\quad \quad \quad a(1, n-1) \ a(n-1, n)$

Summary of CYK

- Grammar is in CNF, input is $x = x_1 x_2 \dots x_n$.
- **For** $r = 1$ to n // diagonal
 $a(r, r) = \{B \mid B \sqsupseteq x_r\}$
- **For** $d = 2$ to n // super-diagonals
 For $r = 1$ to $n-d+1$ // row
 $c = r+d-1$; // column
 $a(r, c) = \emptyset$; // entry to compute
 For $k = r$ to $c-1$
 For each production $B \sqsupseteq CD$
 If $C \sqsupseteq a(r, k)$
 and $D \sqsupseteq a(k+1, c)$
 add B to $a(r, c)$;
- x is in the language iff $S \sqsupseteq a(n, n)$.

Verification by "Pedestrian Simulation"

```
main()
{
  int n = 5;
  int r, c, d, k;

  for( d = 2; d <= n; d++)
  {
    std::cout << "-----" << std::endl;
    std::cout << "diagonal = " << d << std::endl;
    for( r = 1; r <= n-d+1; r++)
    {
      c = r + d - 1;

      std::cout << std::endl;
      std::cout << "To compute a(" << r << ", " << c << ") use: " << std::endl;

      for( k = r; k < c; k++)
      {
        std::cout << "a(" << r << ", " << k << ") paired with "
          << "a(" << (k+1) << ", " << c << ")" << std::endl;
      }
    }
  }
}
```

Verification by "Pedestrian Simulation"

diagonal = 3

To compute $a(1, 3)$ use:
 $a(1, 1)$ paired with $a(2, 3)$
 $a(1, 2)$ paired with $a(3, 3)$

To compute $a(2, 4)$ use:
 $a(2, 2)$ paired with $a(3, 4)$
 $a(2, 3)$ paired with $a(4, 4)$

To compute $a(3, 5)$ use:
 $a(3, 3)$ paired with $a(4, 5)$
 $a(3, 4)$ paired with $a(5, 5)$

diagonal = 4

To compute $a(1, 4)$ use:
 $a(1, 1)$ paired with $a(2, 4)$
 $a(1, 2)$ paired with $a(3, 4)$
 $a(1, 3)$ paired with $a(4, 4)$

To compute $a(2, 5)$ use:
 $a(2, 2)$ paired with $a(3, 5)$
 $a(2, 3)$ paired with $a(4, 5)$
 $a(2, 4)$ paired with $a(5, 5)$

Additional Things to Ponder

- Simplify indexing expressions, e.g. by using transpose of matrix
- Recover derivation tree(s) from the algorithm



Real-World Parsing

- The value of CYK is an upper-bound for **arbitrary** context-free languages.
- $O(n^3)$ is too slow for programming languages, where $O(n)$ is desired (and doable for suitably-deterministic languages).
- May be acceptable for natural languages or other special languages.



Non-PL Applications of Parsing

- Natural Language
- Genomics (RNA strings)
 - Stochastic Context-Free Grammars
 - Transformational Grammars (transform one string to another)
- Art
 - Understanding a piece of art

Aside: L-Systems

- With grammars, productions are applied sequentially at any site in the sentential form where the LHS matches.
- With L-systems, productions are applied at all sites simultaneously, sort of like parallel processing or cellular automata.

Example: Fibonacci L-System

- Production
 - $a \rightarrow b$
 - $b \rightarrow ba$
- Derivation
 - a
 - b
 - ba
 - bab
 - babba
 - babbabab
 - babbababbabba
 - ...

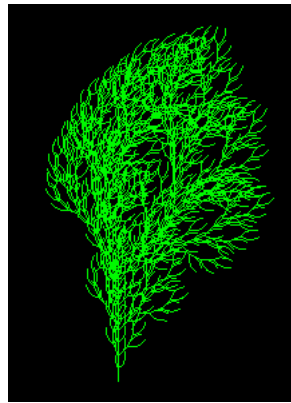
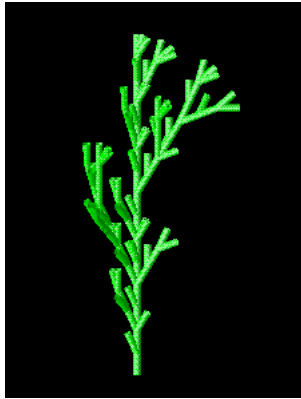
Example: Thue Sequence L-System

- Production
 - $a \rightarrow ab$
 - $b \rightarrow ba$
- Derivation
 - ab
 - abba
 - abbabaab
 - abbabaabbaababba
 - abbabaabbaababbaababbaabbabaab
 - ...
- Distinctions
 - Self-similar (fractal) nature.
 - There is a limit infinite sequence containing all strings as prefixes.
 - No string has a subsequence of the form xxx.

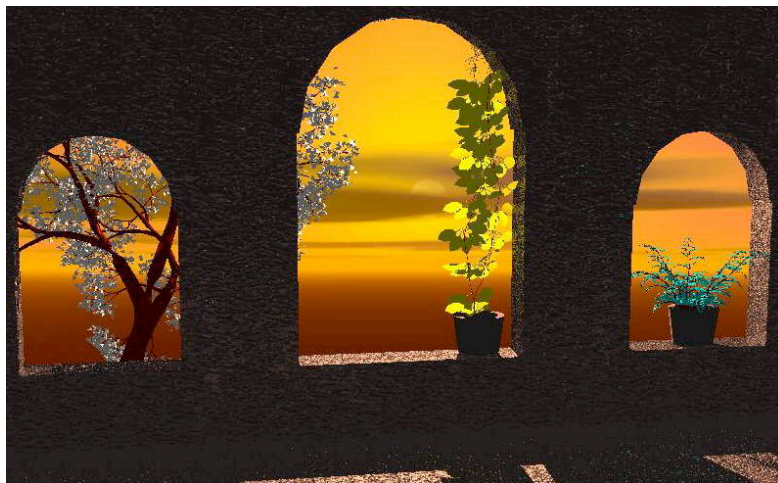
Applications of L-Systems?

- Studied extensively as models for biological development (particularly plants)
- Letters in L sequence can be interpreted as commands to add features, something like "turtle graphics"

Plants constructed using L-Systems



L-System Garden





More Examples:

- Przemyslaw Prusinkiewicz, The Algorithmic Beauty of Plants, Springer-Verlag, 1990.